Programmation orientée objet. Cours 1

Marie-Pierre Béal et Carine Pivoteau
UGE BUT 1

Programmation objet en Java. Records et objets.

Bref historique des langages de programmation

Langages

- Dans les années 60 : COBOL, FORTRAN, LISP, ALGOL
- Langages impératifs et structurés (années 70) : C, Pascal
- Langages fonctionnels (années 80 et 90) : ML, OCaml, Haskell
- Langages orientés objets (années 80 et 90) : Smalltalk, C++, Objective C,
- Langages multiparadigmes : Java, Python

Styles de programmation

Style impératif versus fonctionnel

- **Un langage impératif** :
  - exécute des commandes
  - modifie un état (une case mémoire)
- **Un langage fonctionnel** :
  - exécute des fonctions
  - la valeur de retour d'une fonction ne dépend que des valeurs des paramètres

Style objet

- Un programme est vu comme
  - une communauté de modules autonomes (objets)
  - disposant de leurs ressources propres
  - et de moyens d'interaction.
- Utilise des *classes* pour décrire les structures et leur comportement.
- Usage intensif des **relations entre les objets**.
- Langages typiques : C++, Java, Ruby, Python, C#...

Le langage Java

Java est un langage

- typé statiquement
- multi-paradigme (impératif, fonctionnel, orienté objet, générique, déclaratif et réflexif)
- avec encapsulation, sous-typage, liaison tardive

Histoire de Java

Versions de Java

- Java 1.0 (1995), Orienté objet
- Java 1.5 (2004), Types paramétrés
- Java 1.8 (2014), Lambda
- Java 17 (2021), Record + Sealed types
- Java 21 (2023) Pattern Matching
- Java 23 actuel, JEP 477 Implicitly Declared Classes and Instance Main Methods, JEP 476 Module Import Declarations

créé par James Gosling, Guy Steele et Bill Joy à SUN Microsystem en 1995. Il est basé sur C (syntaxe) et Smalltalk (machine virtuelle).

Open source depuis 2006 `http://github.com/openjdk/jdk`

Démarrer en Java : Hello World

- On écrit le code de la *classe* `Hello` dans un fichier `Hello`**.java** (attention à la majuscule !) :

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

- On compile (dans un terminal) avec la **commande** `javac`

```
$ javac Hello.java
$ ls
Hello.class Hello.java
```

- On lance l'exécution (l'interprétation du bytecode) par la machine virtuelle Java avec la commande `java`

```
$ java Hello
Hello World!
```

Autre exemple

- On écrit le code dans le fichier `AdditionTest.java` :

```
public class AdditionTest {
  public static void main (String[] args) {
    var n = 100;
    var m = 200;
    System.out.println(n + m);
  }
}
```

Important : le code écrit en Java doit toujours se trouver à l'intérieur d'une classe (ou d'un record ou d'une interface).

- On compile avec `javac AdditionTest.java`
- On lance l'exécution avec `java AdditionTest`, on obtient 300.

Expressions et types

Toute expression (portion de code que l'on peut évaluer) a une valeur et un type. Les valeurs peuvent être :

- des valeurs **primitives** (entiers, booléens, caractères, ...)
- des **références (pointeurs) à des objets** (ou des tableaux, mais nous verrons ça plus tard).

Les **types primitifs** sont :

`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`.

Les **objets** peuvent être vus comme des types plus "sophistiqués" déjà présents dans le langage (par ex. `String`) ou créés par le programmeur.
Remarque : les types primitifs commencent tous par des minuscules.
Par convention, les types objets doivent tous commencer par des majuscules !

Variables

Une *variable* est le nom d'un emplacement mémoire qui peut contenir une valeur. Son *type* décrit la nature des valeurs qu'elle peut contenir.

- Si le type est un type primitif, la valeur est de ce type.
- Si le type correspond à un objet, la valeur est une référence à un objet de ce type (ou d'un sous-type, voir cours suivants).

```
public class Test {
  public static void main(String[] args) {
    var i = 1;
    String word = "mot"; // on peut mettre var word
    System.out.println(word + " " + i);
  }
}
```

En Java, un objet ne peut être manipulé que par une référence.

Les objets

Un objet (ou instance)

- est un module autonome,
- qui a ses propres ressources (ses champs),
- et des actions qu'il peut effectuer (ses méthodes).

Pour **utiliser** un objet, il suffit de connaître ses méthodes (par exemple, pour utiliser une `String`, on peut consulter la doc de ses méthodes : `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/String.html`)

Records

Depuis Java 14, il est possible de définir simplement ses propres objets comme des records (dans la suite du cours, nous verrons que l'ont peut également les définir comme des *classes*).

Record

- Un record est un **tuple nommé non modifiable**.

Autrement dit :

- C'est un *enregistrement* de données.
- Ces données sont stockées dans des **champs** qui portent des noms.
- Les valeurs des champs ne peuvent **pas être modifiées**.

Similaires aux *tuples* de Python et aux *struct* du C.

Définir et créer de nouveaux records

Pour définir un nouveau record, il suffit de

- donner son nom,
- et les données qui le constituent, en indiquant leur type.

Dans le fichier `Point.java`, on écrit la définition du record `Point` :

```
public record Point(int x, int y) { }
```

Chaque record est écrit dans son propre fichier `.java` qui porte obligatoirement le même nom que le record. Et ce nom doit commencer par une majuscule.

On peut ensuite créer des points en utilisant le mot-clé `new` et en choisissant les valeurs de `x` et `y` :

```
public class Test {
  public static void main(String[] args) {
    var point1 = new Point(5, 6);
    var point2 = new Point(-1, 3);
  }
}
```

Exemple d'utilisation

```java
public class Test {
  public static void main(String[] args) {
    var point1 = new Point(5, 6);
    var x = point1.x(); // appel de l'accesseur du champs x
    var y = point1.y(); // appel de l'accesseur du champs y
    System.out.println("(" + x + " " + y + ")"); // (5, 6)
    System.out.println(point1.toString()); // Point[x=5, y=6]
    var point2 = new Point(x, y);
    System.out.println(point2.equals(point1)); // true
    System.out.println(point2.equals(new Point(8, 4))); // false
    System.out.println(point2); // appelle toString()
    // Point[x=5, y=6]
  }
}
```

Records : constructeur et méthodes

Pour chaque nouveau record, le langage fournit (entre autres) :

- Un constructeur canonique permettant de créer des objets de ce type avec `new` et en indiquant les valeurs des champs.
- Les accesseurs pour chaque champs (qui portent le même nom). Chaque accesseur renvoie la valeur du champs du record sur lequel on l'appelle. Attention, cette valeur est typée comme le champs.
- Une méthode `toString` qui permet d'afficher le record. Plus précisément, la méthode renvoie une chaîne de caractères (`String`) qui représente ce record.
- Une méthode `equals` qui permet de tester l'égalité. Elle prend en paramètre un autre record et renvoie le booléen `true` s'il est égal au record sur lequel on l'appelle, et `false` sinon.

On appelle une méthode sur un objet avec le point (symbole $\cdot$).

Le test d'égalité

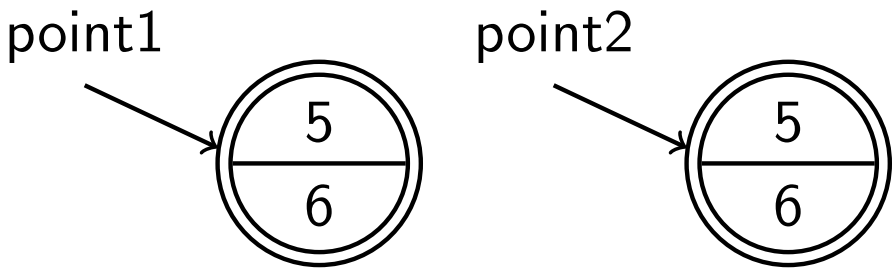
Pour **tester l'égalité des objets, on doit obligatoirement utiliser la méthode** `equals` qui teste l'égalité "structurelle", c'est à dire l'égalité champs par champs.
Si un champs est un objet, l'égalité est testée en utilisant `equals sur ce champs` (sinon, c'est un primitif, on utilise ==).

```
public class Test {
  public static void main(String[] args) {
    var point1 = new Point(5, 6);
    var point2 = new Point(5, 6);
    var point3 = new Point(6, 5);
    var point4 = point1;
    System.out.println(point2.equals(point1)); // true
    System.out.println(point1.equals(point2)); // true
    System.out.println(point1.equals(point3)); // false
    System.out.println(point1.equals(point4)); // true
  }
}
```

Les objets et leurs références

Un objet est un module situé dans une zone mémoire.
On le manipule par son adresse appelée **référence** (ou pointeur, mais sans arithmétique).

```
public class Test {
  public static void main(String[] args) {
    var point1 = new Point(5, 6);
    var point2 = new Point(5, 6);
  }
}
```

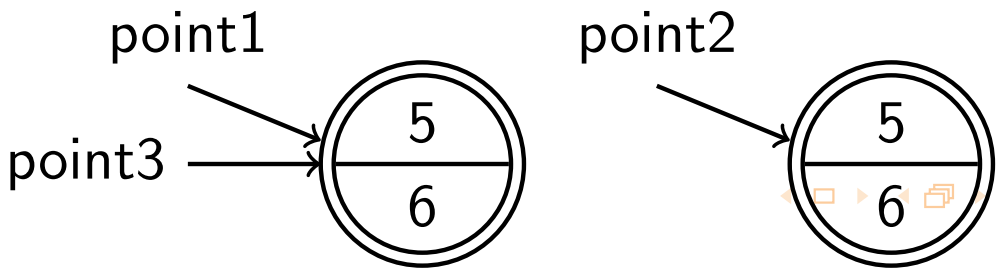


En java, **il existe une référence spéciale** `null`**.** On peut l'utiliser comme "non" valeur de n'importe quelle variable d'un type objet. Dans ce cas, il n'y a pas de zone mémoire réservée.

L'opérateur ==

- Il sert à tester l'égalité des primitifs.
- Sur des objets, l'opérateur == permet uniquement de tester l'égalité des références (c'est à dire, l'endroit ou pointe la flèche).

```
public class Test {
  public static void main(String[] args) {
    var point1 = new Point(5, 6);
    var point2 = new Point(5, 6);
    var point3 = point1;
    System.out.println(point2.equals(point1)); // true
    System.out.println(point2 == point1); // false
    System.out.println(point3 == point1); // true
  }
}
```



D'autres méthodes dans les records ?

On peut ajouter des méthodes propres au record à l'intérieur de sa définition, pour définir de nouvelles actions.
L'en-tête d'une méthode est toujours de la forme suivante :

```
typeDeRetour nomMethode (type1 param1, type2 param2, ...)
```

Par exemple, on peut demander à un `Point` peut s'il est au dessus d'une ligne horizontale :

```
public record Point(int x, int y) {
  public boolean isAbove(int horizontalLine) {
    return y > horizontalLine;
  }
}
```

```
...
var point = new Point(-5, 6);
System.out.println(point.isAbove(4)); // true
System.out.println(point.isAbove(10)); // false
...
```

Autres exemples

Un point peut renvoyer son opposé :

```
public record Point(int x, int y) {
  public Point opposite() {
    return new Point(-x, -y);
  }
```

ou calculer à quelle distance il se trouve d'un autre point :

```
  public double distance(Point other) {
    var xDiff = x - other.x;
    var yDiff = y - other.y;
    return Math.sqrt(xDiff * xDiff + yDiff * yDiff);
  }
```

```java
public class Test {
  public static void main(String[] args) {
    var point = new Point(-5, 6);
    var point2= point.opposite();
    System.out.println(point2); // Point[x=5, y=-6]
    System.out.println(point2.distance(point));
    // 15.620499351813308
  }
}
```